Design for Verification with Dynamic Assertions*

Peter C. Mehlitz
Computer Sciences Corporation, NASA Ames Research Center
pcmehlitz@email.arc.nasa.gov

Abstract

Completed design and implementation are often regarded as pre-requisites of any verification. While recent development methods establish testability as a design criterion, there is no corresponding design support for other verification methods like model checking and static analysis. Since these methods have inherent scalability problems, their application becomes more difficult where it is most needed - for complex systems.

Our *Design-for-Verification* (D4V) approach attempts to close this gap using a variety of techniques, such as design patterns, APIs and source annotations. This paper presents a overview of D4V, and introduces *Dynamic Assertions* as one of the proposed D4V techniques.

Dynamic Assertions are dedicated, non-intrusive check objects that are dynamically activated, evaluated and deactivated via assertions of their target objects. Since these check objects can have their own state, they can be used to verify a broad range of properties. Properties can be expressed in the target programming language, and checked in a testing environment. In addition, *Dynamic Assertions* can be configured via call contexts, making them suitable for connector-specific verification of component based systems.

1 Introduction

Verification traditionally has been regarded as a last pre-delivery development phase to show that a certain system implementation meets its requirements specifications. This perspective can fall short of practical system development needs in two ways:

(A) Development process related - Requirement specifications of complex systems are often incomplete, resulting in a need to validate the system - or artifacts thereof - as soon as possible. If validation cannot be started before most of the system has been implemented and verified, such a "proven to be correct" implementation of the wrong requirements might not only be useless, but is also the most expensive way to fail.

(B) Verification technology related - The more complex a system is, the harder it usually is to find its malfunctions. Due to potentially large input data sets and hard-to-control environment dependencies (e.g., thread scheduling), conventional testing can often only achieve limited confidence in correct system behavior. Formal, automated verification techniques like software model checking [4] can be useful to overcome this restriction, but are usually exponential with respect to system complexity. If these verification tools cannot be applied before the system reaches a critical level of complexity, they are not applicable at all without expensive and error-prone modeling.

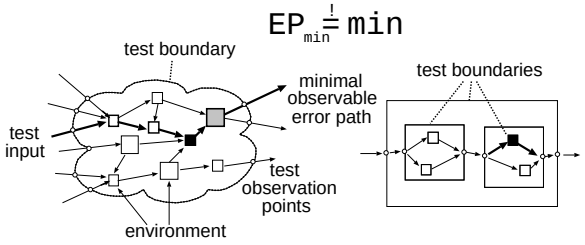
The consequence of the first deficiency is that more contemporary development process models [8] and methodologies [5] put the emphasis on incremental development with rapid turn-around, to achieve user feedback based on design-simulations and runnable artifacts (tests) early in the development process. While this implicitly requires a certain modularity, e.g. to enable unit tests, there are usually no verification-specific design guidelines or rules to directly support formal verification methods.

So far, most of the effort to overcome verification technology related deficiencies has been spent on improving verification tools, but the achieved gains have

*The research described in this report was performed at NASA Ames Research Center's Automated Software Engineering group, and is funded by NASA's Engineering for Complex Systems (ECS) program

been nullified by rapidly growing system complexities, due to increased computational resources and widespread use of ever growing frameworks. For example, the size of the Java system framework libraries has been expanded by more than a factor of 20 since its first release in 1997. It therefore seems unlikely that a tool-centric verification approach alone will succeed in conquering complexity.

The importance of a suitable design is widely accepted in testing, e.g., to ensure minimal observable error paths, but there still is a lack of specific verification-oriented design measures.



This is all the more unsatisfactory since it is well known that software systems usually have a large state space, and even small program changes can result in huge differences regarding the verification-relevant number of states.

2 D4V

Our *Design for Verification* (D4V) approach tries to overcome the scalability problems of formal verification methods by turning verifiability into a explicit design criterion. From a verification point of view, systems are no longer regarded as black boxes, but are assembled from, and annotated with, D4V specific components based on properties that later-on can be checked.

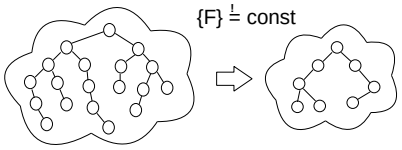
A major hypothesis of D4V is that the same design principles that are considered to be best practices from a general software engineering perspective (e.g. usage of *Design Patterns* [1]) can also be successfully employed for automated verification. Separation of concerns and model based abstractions work well for humans and machines.

There are three major categories of D4V techniques.

2.1 General State Space Reduction

Measures in this category do not change the functionality of the system, but try to implement it in a way that

- minimizes accidental complexity [2]
- enables the use of existing verification tools



So far, there are two sub-categories, both aiming at improved software model checking capabilities.

2.1.1 Reduction of Concurrency

Many system designs use concurrency to separate seemingly independent computations, just to find out in the implementation phase (or subsequent extensions) that some shared resources do require synchronization (e.g. global services like memory allocation). This overlayed synchronization is often error prone (liveness and safety properties), and can dramatically increase the number of relevant states for model checking, which ideally has to consider all potential thread interleavings.

Certain standard design patterns can be used to centralize synchronization in library components, and eliminate the need for application specific inter-thread synchronization (e.g. message passing via centralized queues). In addition, these patterns can sequentialize processing while maintaining good code separation, and therefore significantly reduce the state space a model checker has to look at.

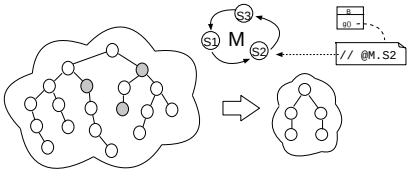
2.1.2 State Space Closing Abstractions

The explicit usage of counters and time values in highly repetitive systems imposes a serious problem for software model checkers. Since these variable values can differ for each cycle, the state space is effectively multiplied by the maximum values of the corresponding variable types, in all likeliness exceeding the memory constraints of the checker if defects cannot be found in early cycles.

D4V addresses this problem by a set of state closing abstraction constructs. Typically, counters and timers are used to check for value differences (e.g. as a progress monitor), or to produce logging output. From a model checkers perspective, the first usage can be abstracted into a simple non-deterministic choice (difference lower or bigger than a certain threshold), the second one usually can be ignored. By using the D4V abstractions instead of primitive type variables, the implementation can be easily exchanged between runtime/testing- and verification-environments.

2.2 Design Model Preservation and Extraction

Design and implementation are always based on a number of models. Nobody starts to write code for complex systems without an abstraction of the process model, potential extensions, or underlying data models.



Unfortunately, these models are often either informal, not kept up-to-date, or simply lost in subsequent implementation details, requiring expensive and lossy restoration from application sources during system verification. D4V tries to avoid this problem by means of preserving the relevant specifications "in-source", using code annotations or specific APIs. There are three major models in the D4V context.

2.2.1 Check Points

This model refers to the correctness aspect of a system, and uses annotations and standard assertions for its implementation to mark locations where the system needs to be in a consistent state. This is based on the observation that - with respect to most functional properties - the state space from a software model checkers point of view mainly consists of temporary states, and only certain fixpoints need to be verified (e.g., before results are distributed). *Check Points* mark these locations, specify the relevant assertions, and link them back to corresponding requirements.

2.2.2 Control Points

Control Points capture the dynamic process model of concurrent systems, with their major synchronization and communication dependencies. They can be used to hint software model checkers with potential atomic sections for partial order reduction (i.e. reducing the number of thread interleavings), defect-type- specific scheduling strategies, or to generate simplified process models (like labeled transition systems) for use in dedicated tools (e.g. LTSA [6]). Control Points can either be specified with special APIs, thereby avoiding redundancy and inconsistency due to inheritance anomaly, or with code annotations providing a certain level of intended redundancy for consistency checks.

2.2.3 Extension Points

During their lifecycle, complex systems usually are subject to significant modifications and extensions. A typical effect is that at some point functional extensions break the original design, violating properties that did hold in previous versions. Therefore, a major task in the specification and design phase is to anticipate future extensions that should be supported, and to mark the corresponding programming constructs (like classes, interfaces, methods, delegation objects) as *Extension Points*. This augments the inheritance control features of the programming language (like access attributes) by semantically grouping related elements according to major extension concepts, linking them back to their specifications, and enabling consistency checks without the need for full fledged static analysis.

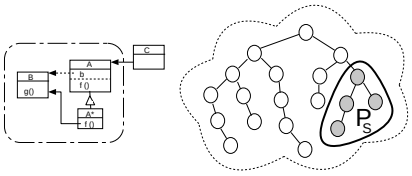
It is important to note that model extraction differs from other D4V categories in that tool generated results (checkable model instances) are *not* functionally equivalent to the original system, they capture only simplified aspects of it. This is most obvious for the Extension Points model, which does not refer to any runtime behavior and is only used to group and link potential source modifications.

2.3 Property Specific State Space Partitioning

In addition to reducing the whole state space, or transforming it into simplified models, a very promising approach is to structure and compose the system so that properties can be checked in subsets of the state space. This can be done in two different ways.

2.3.1 Structure Based Properties

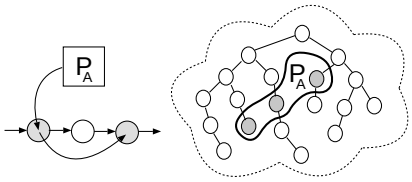
This approach uses specialized design pattern instances that guarantee certain key properties if their corresponding (formal) usage rules are fulfilled, which is mostly achieved by information hiding. Patterns are chosen from application domain specific libraries based on their properties, and users can get feedback regarding verifiable system aspects even before the implementation is started.



Since many design patterns work by means of delegation and abstract interfaces, usage checks will mainly consist of specialized static analysis of call trees inside of interface functions (e.g. making sure certain functions are not used). We assume structure based properties will be the most important D4V measure.

2.3.2 State Annotation

While structure-based properties use specific designs to modify and constrain the verification-relevant state space, this approach starts with an existing design, explicitly identifying and linking states that have to be checked in the context of a certain property. In contrast to local assertion expressions, checks can carry information between the marked states, enabling the use of simple automatons to keep track of evaluation results (e.g. for protocol verification).



We start with a check object, and then attach it to relevant program states. Like Check Points, this mainly aims at functional properties that should be verifiable in testing and model checking environments.

In the second part of this paper, we present *Dynamic Assertions* as one example of this D4V category.

3 Dynamic Assertions

The basic idea behind this state annotation technique is to use the assertion mechanism of a programming language to delegate property verification in specific states to dedicated, dynamically created check objects. The goal is to ease property specification and shrink the property relevant state space. The requirements are that checks can be

- utilized on a per-object basis
- explicitly enabled and disabled in specific states
- formulated with programming language expressions, and a low level of theory- or implementation- inflicted overhead
- conditionally compiled (i.e., do not introduce production system runtime overhead or other side effects)

3.1 Example

While dynamic assertions are mostly programming language and runtime-environment independent, there are several levels of support that can be used to implement them. The following examples uses a source code preprocessor (iContract [3]) to implement pre-, post-conditions and invariants [7] for the Java programming language.

Assuming that instances of class `A` can be used in a variety of contexts, we might not be able to specify usage-specific properties as invariants of `A`.

```
/** @invariant DynAssertion.check(this) */
class A {
  int x; ..
  public void setX (int x) { this.x = x; }
}
```

This defines a class that automatically evaluates the expression `DynAssertion.check(this)` as a invariant check, i.e. before and after every method invocation on an `A` instance. The instrumented code generated by the programming-by-contract preprocessor might look like

```
..
public void setX (int x) {
  if(!DynAssertion.check(this))
     throw new Exception("precond");
  ..
}
```

`DynAssertion` refers to a library class that is used to manage check objects

```
abstract class DynAssertion {
  static Map assertions;

  public static boolean check (Object o){
    Object da = assertions.get(o);
    if (da != null)
      return ((DynAssertion)da).check();
    else return true;
  }

  public static add (Object o,
                     DynAssertion da){
    ..assertions.put(o,da);
  }

  public boolean check () {return true;}
}
```

The following class `B` represents one of the users of `A`, whose instances store a (non-aggregate) reference to an `A` object that - potentially - can be modified outside the class `B` context.

Our example property is

> between the execution start of f() and the execution end of g() of a B instance, both y and the related a.x field values have to be greater than zero

which is translated into a `DynAssertion` object that is created and activated (linked to target objects `a` and `b`) as a pre-condition of method `f()`, and removed as a post-condition of method `g()`.

```
/** @invariant DynAssertion.check(this) */
class B {
  A a; .. int y;
  B (A a) { this.a = a; }

  /** @pre DynAssertion.add( this, a,
   *     new DynAssertion ("myCheck") {
   *       public check () {
   *         return (y > 0)&&(a.getX() > 0);
   *       }
   *     })
   */
  void f (..) {..}

  /** @post DynAssertion.remove(this,a,
   *                            "myCheck")
   */
  void g (..) {..}
  ...
}
```

The following code fragment represents a possible usage scenario of this dynamic assertion

```
class C {
  static A a = new A(); // not aware of
                        // being used by 'B'
  void foo() {
    B b = new B(a);

    b.f();     // DA activated
    ...
    a.setX(-1);// violates property
    ...
    b.g();     // DA removed
   ...
  }
}
```

While the semantics of this example are certainly artificial, it is important to note that checks are only performed on the two involved objects (`a` and `b`), only within the specified range of states (between `f()` and `g()` executions), and only at states that potentially can violate the property (method calls on the two involved objects).

3.2 Dynamic Assertions With State

What really sets dynamic assertions apart from normal assertions is the fact that they are objects, being able to store their own state. This can be especially powerful if it is combined with execution environment features.

A simple, yet practical example is a special dynamic assertion class to verify (partial) method call protocols in

Java. Assume that a certain class A has a lot of methods that can be called in any order, except that

> an open() call cannot directly or indirectly be followed by second open() call before close() was called

We can implement this by means of a `DynAssertion` derived class `ProtocolAssertion`, which encapsulates a pattern matcher that keeps track of previous check evaluations (`ProtocolAssertion` uses the Java 1.4.1 stack trace support to identify method calls)

```
/** @invariant DynAssertion.check(this) */
class A {
  /** @pre DynAssertion.add( this,
   *     new ProtocolAssertion("open-close"
   *                   "{~open} close"))
   */
  void open () {..}
  void close () {..}
  void foo () {..}
  ...
}
```

Once the pattern has been successfully matched, the `ProtocolAssertion` instance can automatically remove (de-activate) itself. The property specification is kept close to the location it refers to (execution of `open()` method), and does not require any information that is not readily available in its own context (e.g., no complete state model of all `A` instances). Since the assertion object is associated with a specific target object, it can co-exist with other dynamic assertions.

This example shows how dynamic assertion state can be used for non-trivial safety properties ("no state with consecutive `open()` calls"). By means of a global assertion registry and some program fixpoints (i.e., states which are known to be reached), we can also implement liveness properties like

> any postEvent() execution is eventually followed by its corresponding processEvent() execution

To implement this property, we use the assertion registry as the check memory, not the assertion object itself, and associate the assertion objects with their corresponding event objects, checking the assertion registry for not-yet-removed objects as a post-condition of the `dispatcherLoop()` execution.

```
class EventQueue {
  /** @pre DynAssertion.add( e,
   *     new DynAssertion("posted"){})
   */
  void postEvent (Event e) {..}

  /** @post DynAssertion.remove(e,"posted")
   */
  void processEvent (Event e) {..}

  /** @post DynAssertion.hasNo(Event.class,
   *                           "posted")
   */
  void dispatcherLoop () {
    ..
    while (!done){
      Event e = queue.getNextEvent();
      ..
      processEvent(e);
    }
    /** @controlpoint exitDispatcherLoop */
  }
  ...
}
```

This property could certainly better be verified by means of a D4V design pattern (i.e. as a structure based property), but demonstrates how dynamic assertions can be used in a design that is not property-aware at all. It should be noted that the property would also require application storage (i.e. additional state) if we use a temporal logic formula like $\Box(postEvent \Rightarrow \Diamond processEvent)$ since we have to reason about the (abstract) state of `Event` objects, not the `EventQueue` (`processEvent()` calls for non-posted events do not count).

While suspicious from a model checking point of view, the halting-problem related dependency on program fixpoints is of less concern in a practical context. Liveness checks on application specific fixpoints (*Control Points* from a D4V perspective) are usually first class properties themselves (indicated by the `@controlpoint` annotation in the example). In addition, most runtime environments provide critical system-level functionality related fixpoints like exit hooks and finalizers that have to be guaranteed in order to assume any deterministic system behavior. In a model checker environment, we could also mark certain states in which the assertion registry is checked during a backtrack operation (i.e., if the model checker has explored the whole state space underneath this point).

3.3 Application

The mechanism integrates seamlessly with standard assertions and programming-by-contract support. It does not require any specific runtime environment, and therefore can be used for normal testing. However, assuming that a property violation during a checked sequence is inflicted in a concurrent context (e.g., by calling `a.setX(-1)` from another thread), it is obvious that conventional testing might not be able to detect all potential violations. In this case, we can use a software model checking environment like *Java Pathfinder* [4], to systematically verify many - if not all - possible execution paths.

The model checker can treat dynamic assertions like normal application code, and only has to check for standard program exceptions. While this avoids overhead within the model checker, which does not have to be aware of how properties are tested, it also increases the state space of the application. This is a primary reason why it is important to constrain the lifetime of check objects, and to minimize the number of check- relevant states.

On the other hand, if a model checker is able to intercept dynamic assertion calls, it can use the activation, deactivation and target object information as hints of how to further reduce the application state space (e.g. minimizing the number of thread interleavings by means of partial order reduction). The capability to intercept these calls has been implemented in *Java Pathfinder* as the generic *Model-Java-Interface* (MJI).

Working on a per-instance basis in the context of method invocations, dynamic assertions represent a runtime mechanism that naturally lends itself to component based systems, which can be characterized by

- a high degree of context specific, un-anticipated reuse of components
- a low degree of static usage/compatibility checks (component sources might not be available at all)

Reasoning about components depends on their external interface, which is mapped into specific methods of the component implementation. These methods could be used as the injection points of connector-specific assertions, thus enabling a “wiring”-based verification.

```
/** @invariant DynAssertion.check(this) */
class SomeComponent {
  /** @pre DynAssertion.add( this,
   *     Composition.getConnectorAssertion())
   */
  void externalFunction () {..}
  ...
}
```

This uses a verification-specific repository class `Composition` to look up assertion objects via caller/callee (connector) information provided by the runtime environment.

3.4 Caveats

Dynamic assertions do have their caveats. From a users perspective, the most obvious one is that dynamic assertions should be free of side effects with respect to normal program control flow and data values. The user also has to be aware of that delegating the property verification to dedicated objects is only efficient if there are relatively few target objects that have to be checked in a limited number of states. If a property has to be checked for all instances of a certain type in a large number of states, inlined assertions, contracts, or model-checking specific verification methods like temporal logic should be preferred.

With respect to the implementation of dynamic assertion infrastructure, it is important to avoid memory leaks due to the linkage between assertion- and target-objects, especially since assertion objects can be stored in a global repository (i.e. are reachable from the root set). This can be achieved by weak references, which are available in most contemporary programming environments with automatic memory management (like Java and C#). Another important implementation aspect is to avoid inheritance anomaly. If the invariant or pre- and post- condition instrumentation is not inheritance aware, or dynamic assertions are created in free assertion expressions, derived classes can easily loose property-relevant states, i.e. inheritance can break the assumed correctness model. This can be prevented by means of using existing programming-by-contract systems [3]with their well-defined inheritance behavior (weakening pre-conditions, strengthening post-conditions).

4 Conclusions and Future Work

In this paper, we have sketched out motivation and scope of D4V as a collection of tools and techniques to enable verification of complex systems, and presented *Dynamic Assertions* as one example technique from its repertoire.

The wide range of potential D4V categories makes it obvious that future D4V work will be mainly driven by the needs of real world test-cases, and influenced by current and planned capabilities of our verification toolset [9].

The next step will be the demonstration of a D4V specific design pattern, using techniques like dynamic assertions and model preservation/extraction for its implementation. This will be gradually extended into an application-domain specific pattern system, reflecting our belief that verifiability should be considered as an integral part of the design of complex systems.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Design Patterns Elements of Reusable Object-Oriented Software", Addison Wesley, 1995

[2] Frederick P. Brooks: "No Silver Bullet: Essence and Accidents of Software Engineering", Proceedings of the IFIP '86 conference

[3] Reto Kramer: "iContract - the Java Design by Contract tool" , Proceedings of Technology for Object-Oriented Languages and Systems, TOOLS-USA. IEEE Press, 1998

[4] G. Brat, K. Havelund, S. Park, W. Visser: "Java PathFinder - A second generation of a Java model checker", Proceedings of the Workshop on Advances in Verification, Chicago, Illinois, July 2000.

[5] Kent Beck: "Extreme Programming Explained: embrace change", Addison Wesley. 2000

[6] J. Magee, J. Kramer: "Concurrency: State Models & Java Programs", Wiley 1999

[7] Bertrand Meyer: "Object Oriented Software Construction", Prentice Hall 1997

[8] B. Boehm and P. Bose: "A Collaborative Spiral Software Process Model Based on Theoy W", Third International Conference on the Software Process, 1994.

[9] M. Mansouri-Samani, P. Mehlitz, L. Markosian, O. OMalley, D. Martin, L. Moore, J. Penix, and W. Visser : "Propel: Tools and Methods for Practical Source Code Model Checking", DSN 2003Workshop on Model Checking for Dependable Software-Intensive Systems. San Francisco, June, 2003 (to appear)